C++ Grundlagen

Noah Kirschmann

Hello World C

```c
#include "stdio.h"

int main(){
    char hello[] = "hello";
    int value = 5;

    printf("%s world! %i\n", hello, 5);

    return 0;
}
```

Hello World C++ 1

```cpp
#include <iostream>

int main(){
    char hello[] = "hello";
    int value = 5;

    std::cout << hello << " world! " << value << std::endl;

    return 0;
}
```

Hello World C++ 2

```cpp
#include <iostream>
#include <string>

int main(){
    std::string hello = "hello";
    int value = 5;

    std::cout << hello << " world! " << value << std::endl;

    return 0;
}
```

Hello World C++ 3

```cpp
#include <iostream>
#include <string>

int main(){
    std::string hello = "hello";
    int64_t value = 5;

    std::cout << hello << " world! " << value << std::endl;

    return 0;
}
```

Datentypen

- C and C++ use static data typing.
- An object's declaration determines its static type:
  - int n; // n is "[signed] integer"
  - double d; // d is "double-precision floating point"
  - char *p; // p is "pointer to character"
- An object's static type doesn't change during program execution.
- It doesn't matter what you try to store into it.

Datentypen Definition

- Ein Datentyp ist eine Sammung an Compilezeit Eigenschaften eines Objektes
  - Größe und Anordnung
  - Menge der gültigen Werte
  - Menge der gültigen Operationen

Beispiele Datentypen

- On a typical 32-bit processor, type int has:
  - size and alignment of 4 (bytes)
  - Values from -2147483648 to 2147483647,
  - Inclusive integers only
  - Operations including:

    unary +, -, !, ~, &, ++, --binary =, +, -, *, /,%, <, >, ==, !=, &, |, &&, ||
- Nicht möglich bei int:
  - *i // indirection (as if a pointer)
  - i.m // member selection
  - I() // call (as if a function)

Arten von Datentypen

- Builtin types:

  C: bool, int, long, double, float, int*, long [], int (int), etc.

  C++: int64_t, uint32_t, etc.

- User-defines types:

  C: struct

  C++: class

Structs (C)

```c
#include "stdio.h"

struct vec2{
    float x;
    float y;
};

struct vec2 addVectors(struct vec2 v1, struct vec2 v2){
    struct vec2 res;
    res.x = v1.x + v2.x;
    res.y = v1.y + v2.y;
    return res;
}

void printVector(struct vec2 v){
    printf("(%f, %f)", v.x, v.y);
}

int main(){
    struct vec2 a = {1.0f,2.0f};
    struct vec2 b = {2.0f,1.0f};

    struct vec2 c = addVectors(a,b);

    printVector(c);
    printf("\n");

    return 0;
}
```

Structs (C++)

```cpp
#include <iostream>

struct vec2{
    float x;
    float y;
};

struct vec2 addVectors(struct vec2 v1, struct vec2 v2){
    struct vec2 res;
    res.x = v1.x + v2.x;
    res.y = v1.y + v2.y;
    return res;
}

void printVector(struct vec2 v){
    std::cout << "(" << v.x << ", " << v.y << ")";
}

int main(){
    struct vec2 a = {1.5f,2.0f};
    struct vec2 b = {2.0f,1.0f};

    auto c = addVectors(a,b);

    printVector(c);
    std::cout << std::endl;

    return 0;
}
```

Klassen (C++)

```cpp
#include <iostream>

class vec2{
  public:
    float x;
    float y;
};

vec2 addVectors(vec2 v1, vec2 v2){
    vec2 res;
    res.x = v1.x + v2.x;
    res.y = v1.y + v2.y;
    return res;
}

void printVector(vec2 v){
    std::cout << "(" << v.x << ", " << v.y << ")";
}

int main(){
    vec2 a = {1.5f,2.0f};
    vec2 b = {2.0f,1.0f};

    auto c = addVectors(a,b);

    printVector(c);
    std::cout << std::endl;

    return 0;
}
```

Pointer auf Klassen (C++)

```cpp
vec2 addVectors(vec2* v1, vec2* v2){
    vec2 res;
    res.x = v1->x + v2->x;
    res.y = (*v1).y + (*v2).y;
    return res;
}

void printVector(vec2* v){
    std::cout << "(" << v->x << ", " << v->y << ")";
}

int main(){
    vec2 a = {1.5f,2.0f};
    vec2 b = {2.0f,1.0f};

    auto c = addVectors(&a,&b);

    printVector(&c);
    std::cout << std::endl;

    return 0;
}
```

Das Problem von Pointern

```
void printVector(vec2* v){
    std::cout << "(" << v->x << ", " << v->y << ")";
    std::cout << std::endl;
    v++;
    std::cout << "(" << v->x << ", " << v->y << ")";
    std::cout << std::endl;
    v++;
    std::cout << "(" << v->x << ", " << v->y << ")";
}

int main(){
    vec2 a = {1.5f,2.0f};
    vec2 b = {2.0f,1.0f};

    auto c = addVectors(&a,&b);

    printVector(&a);
    std::cout << std::endl;

    return 0;
}
```

```
noa@noa-ZenBook-UX462DA ~/D/H/b/c++ listings> ./a.out
(1.5, 2)
(2, 1)
(3.5, 3)
```

Referenzen

```cpp
#include <iostream>

class vec2{
  public:
    float x;
    float y;
};

vec2 addVectors(vec2& v1, vec2& v2){
    vec2 res;
    res.x = v1.x + v2.x;
    res.y = v1.y + v2.y;
    return res;
}

void printVector(vec2& v){
    std::cout << "(" << v.x << ", " << v.y << ")";
}

int main(){
    vec2 a = {1.5f,2.0f};
    vec2 b = {2.0f,1.0f};

    auto c = addVectors(a,b);

    printVector(c);
    std::cout << std::endl;

    return 0;
}
```

Konstante Referenzen

```
vec2 addVectors(const vec2& v1, const vec2& v2){
    vec2 res;
    res.x = v1.x + v2.x;
    res.y = v1.y + v2.y;
    return res;
}

void printVector(const vec2& v){
    std::cout << "(" << v.x << ", " << v.y << ")";
}
```

Pointer vs Referenz vs Konstante Referenz

- Pointer:
  - Zeiger auf Adresse
  - Vorteil: Schneller Zugriff ohne Kopie
  - Nachteil: Sehr unsicher und fehleranfällig
- Referenz:
  - Zeiger auf Variable
  - Vorteile gegenüber Pointer:
    - Zeigt auf gültige Variable (nie NULL bzw. nullptr)
    - Gleiche Syntax wie Kopie
    - Verhindert Zugiff auf den Arbeitsspeicher
- Konstante Referenz:
  - Wie Referenz nur read-only

Memberfunktionen (Methoden)

```cpp
#include <iostream>

class vec2{
  public:
    float x;
    float y;

    void print() const{
        std::cout << "(" << x << ", " << y << ")";
    }

    vec2 operator+(const vec2& other){
        vec2 res;
        res.x = x + other.x;
        res.y = y + other.y;
        return res;
    }
};

int main(){
    vec2 a = {1.5f,2.0f};
    vec2 b = {2.0f,1.0f};

    auto c = a + b;

    c.print();
    std::cout << std::endl;

    return 0;
}
```

Überladen von Funktionen

Überladen heißt, dass die Funktion abhängig von den Parametern etwas anderes macht. Zum überladen müssen die Parameter in der Länge oder im Datentyp unterscheiden.

Beispiele:

vector * Zahl → Skalaieren
vector * vector → Skalarprodukt

cout « float → float ausgeben
cout « vector → vector ausgeben

Die folgenden Operatoren können in C++ NICHT überladen werden:

:: .* . ?:

```cpp
#include <iostream>

class vec2{
  public:
    float x;
    float y;

    vec2 operator+(const vec2& other) const{
        vec2 res;
        res.x = x + other.x;
        res.y = y + other.y;
        return res;
    }

    vec2 operator*(float scalar) const{
        vec2 res;
        res.x = x * scalar;
        res.y = y * scalar;
        return res;
    }

    float operator*(const vec2& other) const{
        float res;
        res = x * other.x;
        res += y * other.y;
        return res;
    }
};

std::ostream &operator<<(std::ostream &os, const vec2& m) {
    return os << "(" << m.x << ", " << m.y << ")";
}

int main(){
    vec2 a = {1.5f,2.0f};
    vec2 b = {2.0f,1.0f};

    std::cout << a*2.0f << std::endl;
    std::cout << a*b << std::endl;

    return 0;
}
```

Funktionen in C++

Alles was aussieht wie eine Funktion ist eine Funktion. Es kann zwischen allgemeinen Funktionen, Memberfunktionen, Funktionsobjekten (Klassen mit überladenenem () Operator) und Lambda-Funktionen unerschieden werden.

Beispiele:

- Foo.bar()
- Foo()

  Foo kann hier entweder eine Variable oder eine Funktion sein, da es wie eine Funktion verwendet werden kann, ist es eine Funktion.

Klassen Constructor

Der Constructor ist die Funktion, welche beim erstellen der Variable aufgerufen wird.

Wie alle Funktionen kann diese Überladen werden.

```
    float operator*(const vec2& other) const{
        float res;
        res = x * other.x;
        res += y * other.y;
        return res;
    }

    vec2():vec2(0.0f, 0.0f) {};
    vec2(float scalar):vec2(scalar, scalar) {};
    vec2(float X, float Y): x{X}, y{Y} {};
};

std::ostream &operator<<(std::ostream &os, const vec2& m) {
    return os << "(" << m.x << ", " << m.y << ")";
}

int main(){
    vec2 a{1.5f};
    vec2 b = vec2(2.0f,1.0f);

    std::cout << a*2.0f << std::endl;
    std::cout << a*b << std::endl;

    return 0;
}
```

Arrays in C

```c
#include "stdio.h"

void hello(char world[]){
    printf("hello %s!\n", world);
}

int main(){
    char world[] = "world";
    hello(world);

    return 0;
}
```

Das Problem von C Arrays

```c
#include "stdio.h"

void calc(char data[], int size){
    for(unsigned int i = 0; i < size;i++){
        printf("value %i is %c\n", i, data[i]);
    }
}

int main(){
    int value = 20;
    char secret[] = "Some secret text!";
    char dat[] = {'a','b','c','d'};
    calc(dat,value);

    return 0;
}
```

Ausgabe des Programms

```
value 0 is a
value 1 is b
value 2 is c
value 3 is d
value 4 is S
value 5 is o
value 6 is m
value 7 is e
value 8 is
value 9 is s
value 10 is e
value 11 is c
value 12 is r
value 13 is e
value 14 is t
value 15 is
value 16 is t
value 17 is e
value 18 is x
value 19 is t
```

Array print template

```cpp
#include <iostream>

template<typename T, size_t SIZE>
void array_print(const T (&data)[SIZE]){
    for(unsigned int i = 0; i < SIZE-1;i++){
        std::cout << data[i] << ", ";
    }

    std::cout << data[SIZE-1] << std::endl;
}

int main(){
    float d1[] = {1.0,2.0,3.0};
    int64_t d2[] = {4,5,6,7,8,9};

    array_print(d1);
    array_print(d2);

    return 0;
}
```

Eigener Array Datentyp

- Überprüfung der Grenzen
- Kopierbar
- Pointer Möglich
- Alle Datentypen möglich

  → Template Klasse

Array Klasse 1

```
3   template<size_t SIZE>
4   class char_array{
5   private:
6       char data[SIZE];
7   public:
8       char_array(char new_data[SIZE]){
9           for(size_t i = 0; i < SIZE;i++){
10              data[i] = new_data[i];
11          }
12      }
13
14      char_array(const char (&new_data)[SIZE]){
15          for(size_t i = 0; i < SIZE;i++){
16              data[i] = new_data[i];
17          }
18      }
19
20      char& operator[](size_t index){
21          index = index%SIZE;
22          return data[index];
23      }
24  };
25
```

Array Klasse 1

```
26  template<size_t SIZE>
27  void hello(char_array<SIZE> world){
28      for(unsigned int i = 0; i < 20;i++){
29          std::cout << world[i];
30      }
31  }
32
33  int main(){
34      char_array<5> world{{'w','o','r','l','d'}};
35      hello(world);
36
37      return 0;
38  }
```

Ausgabe: worldworldworldworld

Array Klasse 2

```cpp
#include <iostream>

template<typename T, size_t SIZE>
class array{
private:
    T data[SIZE];
public:
    array(T new_data[SIZE]){
        for(size_t i = 0; i < SIZE;i++){
            data[i] = new_data[i];
        }
    }

    array(const T (&new_data)[SIZE]){
        for(size_t i = 0; i < SIZE;i++){
            data[i] = new_data[i];
        }
    }

    T& operator[](size_t index){
        index = index%SIZE;
        return data[index];
    }
};

template<typename T, size_t SIZE>
void hello(array<T, SIZE> world){
    for(unsigned int i = 0; i < 20;i++){
        std::cout << world[i];
    }
}

int main(){
    array<char, 5> world{{'w','o','r','l','d'}};
    hello(world);
    std::cout << std::endl;

    return 0;
}
```

Array Klasse 3

```cpp
#include <iostream>
#include <array>

template<typename T, size_t SIZE>
void hello(std::array<T, SIZE> world){
    for(auto x: world){
        std::cout << x;
    }
}

int main(){

    std::array<char, 5> world{{'w','o','r','l','d'}};
    hello(world);
    std::cout << std::endl;

    return 0;
}
```

Array Klasse 4

```cpp
#include <iostream>
#include <vector>

template<typename T>
void hello(std::vector<T> world){
    for(auto x: world){
        std::cout << x;
    }
}

int main(){

    std::vector<char> world{{'w','o','r','l','d'}};
    hello(world);
    std::cout << std::endl;

    return 0;
}
```